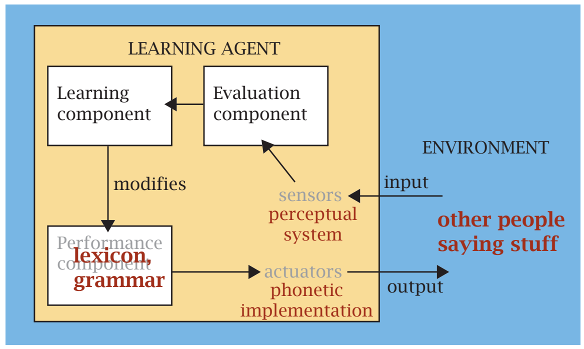
24.964 Fall 2004
Modeling phonological learning

A. Albright
16 Sept, 2004

Class 2: Simple learning models

(1) Learning phonology



(2) Supervised learning

- Learner is given stimuli (inputs) and also answers (outputs)
- Comparing the input and the output lets the learner see what it needs to learn
- Task is to learn a function converting inputs to their corresponding outputs

Unsupervised learning

- Learner receives only input, but no output values
- Model is not told “what to do”
- It looks at the data and tries to find patterns; figure out what types of inputs are likely to occur

(3) Error-driven learning

- *Error rate* = (number of errors / number of cases)
- (1 − error rate) = *accuracy*, or *coverage* of the hypothesis

(4) Distinguishing between different types of errors

	Class Positive	Class Negative
Prediction Positive	True Pos	False Pos
Prediction Negative	False Neg	True Neg

- Correct applications: true positive, true negative
- Misclassifications: false positive, false negative

(See `hepburn3.pl`; this program attempts to calculate false positives and false negatives, but can’t quite do it accurately—why not?)

(5) `hepburn4.pl`

```
$input_file = "Japanese-ToConvert.txt";
open (INFILE, $input_file) or die "Warning! Can’t open input file: $!\n";
$check_file = "Japanese-Converted.txt";
open (CHECKFILE, $check_file) or die "Warning! Can’t open check file: $!\n";

# Construct an array of arrays.
```

```
# The syntax is unintuitive; the square brackets take the list inside of them,
# bundle them up, and store them somewhere.  The first item in the @rules array,
# then, is a reminder of where to go to find those values.
@rules = (
    ["hu", "fu"],
    ["ty", "ch"],
    ["sy", "sh"],
    ["zy", "j"],
    ["ti", "chi"],
    ["si", "shi"],
    ["zi", "ji"],
    ["tu", "tsu"],
);

while ($line = <INFILE>) {
    chomp($line);
    $original = $line;

    for ($i = 0; $i <= $#rules; $i++) {
        $line =~ s/$rules[$i][0]/$rules[$i][1]/g;
    }

    print "$line";

    # Now check answer against the "real" answer in the checkfile
    $check_line = <CHECKFILE>;
    chomp($check_line);
    if ($line eq $check_line) {
        print "\t(correct)\n";
    } else {
        if ($check_line eq $original) {
            # We changed something that we shouldn't have
            print "\t(incorrect - accidentally modified <$original> when we shouldn't have\n";
        } else {
            print "\t(incorrect - need to learn something to change <$original> to <$check_line> \n";
        }
    }
}
```

(6) hepburn5.pl

```
# hepburn5.pl
# Converts Japanese text in "official" Monbushoo (Kunrei-shiki) romanization
#  to more common "Hepburn"-style romanization
# A list of differences can be found at:
#   http://en.wikipedia.org/wiki/Romaji

$input_file = "Japanese-ToConvert.txt";
open (INFILE, $input_file) or die "Warning! Can't open input file: $!\n";
$check_file = "Japanese-Converted.txt";
open (CHECKFILE, $check_file) or die "Warning! Can't open check file: $!\n";

$rules_file = "JapaneseRules.txt";
open (RULESFILE, $rules_file) or die "Warning! Can't open rule file: $!\n";

# Read in the file and store each line in the rules array of arrays
while ($line = <RULESFILE>) {
    chomp($line);
    ($kunrei, $hepburn) = split("\t", $line);
    # Now, place this pair onto the end of the @rules array
```

```
    push(@rules, [ $kunrei, $hepburn ]);
}

while ($line = <INFILE>) {
    chomp($line);
    $original = $line;

    for ($i = 0; $i <= $#rules; $i++) {
        $line =~ s/$rules[$i][0]/$rules[$i][1]/g;
    }


    print "$line";

    # Now check answer against the "real" answer in the checkfile
    $check_line = <CHECKFILE>;
    chomp($check_line);
    if ($line eq $check_line) {
        print "\t(correct)\n";
    } else {
        if ($check_line eq $original) {
            # We changed something that we shouldn't have
            print "\t(incorrect - accidentally modified <$original> when we shouldn't have\n";
        } else {
            print "\t(incorrect - need to learn something to change <$original> to <$check_line> \n";
        }
    }
}
```

(7) `hepburn6.pl` (excerpt)

```
$number_correct = 0;
for ($i = 0; $i <= $#inputs; $i++) {
    # We'll start with the current input, and transform it
    $output = $inputs[$i];
    for ($r = 0; $r <= $#rules; $r++) {
    $output =~ s/$rules[$r][0]/$rules[$r][1]/g;
    }
    print "$output";
    # Now check answer against the "real" answer in the checkfile
    if ($output eq $answers[$i]) {
    print "\t(correct)\n";
    } else {
        if ($answers[$i] eq $inputs[$i]) {
            # We changed something that we shouldn't have
            print "\t(incorrect - accidentally modified <$inputs[$i]> when we shouldn't have\n";
        } else {
            print "\t(incorrect - need to learn something to change <$inputs[$i]> to <$answers[$i]> \n";
        }
    }
}
if ($number_correct == ($#inputs + 1) ) {
    print "\n***Perfect -- all forms accounted for***\n";
}
```

(8) Comparison: decision lists[1]

"All electronics" data set

Age	Income	Student?	Credit	Buys
≤30	high	no	fair	no
≤30	high	no	excellent	no
30-40	high	no	fair	yes
>40	med	no	fair	yes
>40	low	yes	fair	yes
>40	low	yes	excellent	no
30-40	low	yes	excellent	yes
≤30	med	no	fair	no
≤30	low	yes	excellent	yes
>40	med	yes	fair	yes
≤30	med	yes	excellent	yes
30-40	med	no	excellent	yes
30-40	high	yes	fair	yes
>40	med	no	excellent	no

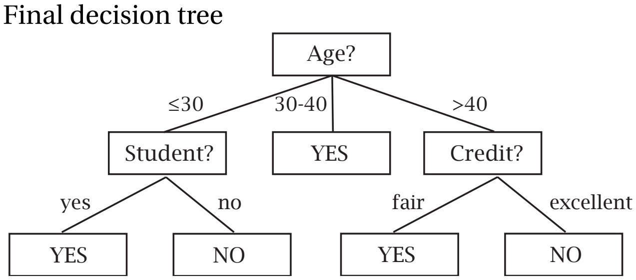
(9) Predictive power of factors (step 1):

Factor	Level	How many buy
Age	≤30	2/5
	30-40	**4/4**
	>40	3/5
Income	low	3/4
	med	4/6
	high	2/4
Student	yes	6/7
	no	3/7
Credit	fair	5/7
	excellent	4/7

(10) Predictive power among remaining cases (step 2):

Factor	Level	≤30 buy	>40 buy
Income	low	**1/1**	1/2
	med	1/2	2/3
	high	**0/2**	0/0
Student	yes	**2/2**	2/3
	no	**0/3**	1/2
Credit	fair	2/3	**3/3**
	excellent	**0/2**	**0/2**

(11) Final decision tree



☞ Why would this approach *not* work for phonology?

[1] http://www.cs.ubc.ca/labs/lci/CIspace/Version3/dTree/index.html

(12) `hepburn7.pl` (excerpt)

```
$iterations = 0;
while ($number_correct != ($#inputs + 1)) {
    $number_correct = 0;
    $iterations++;

    # Try flipping two rules
    $r1 = rand($#rules + 1);
    $r2 = rand($#rules + 1);

    # The following contains an extra fancy bit of code to round of the number when it's printed.
    # Instead of the variables $r1 and $r2, we put a placeholder "%.3f" meaning a floating point
    # (decimal) number with three decimal places.  Then, after the string, we list the variables
    # that should go in those spots (in order)
    printf "Flipping %.3f ($rules[$r1][0]->$rules[$r1][1]) and %.3f ($rules[$r2][0]->$rules[$r2][1])\n", $r1, $r2;
    @rules[$r1, $r2] = @rules[$r2, $r1];

    for ($i = 0; $i <= $#inputs; $i++) {
          # We'll start with the current input, and transform it
          $output = $inputs[$i];
          for ($r = 0; $r <= $#rules; $r++) {
              $output =~ s/$rules[$r][0]/$rules[$r][1]/g;
          }
          # Now check answer against the "real" answer in the checkfile
          if ($output eq $answers[$i]) {
              $number_correct++;
          }
    }
    if ($number_correct == ($#inputs + 1) ) {
          print "\n*** Perfect -- all forms accounted for on iteration $iterations ***\n";
    }
}
```

(13) `hepburn8.pl` (excerpt)

```
# We want to keep a copy of the start state, so we can keep going back to it
for (my $i = 0; $i <= $#rules; $i++) {
          print "keeping original copy of rule $i\n";

      push (@original_rules, @rules->[$i]);
}

for ($t = 1; $t <= $number_of_trials; $t++) {

    # For each trial, we start at the start state and try solving it again
    @rules = undef;

    for (my $i = 0; $i <= $#original_rules; $i++) {
        push (@rules, @original_rules->[$i]);
    }

    $iterations = 0;
    $number_correct = 0;

    while ($number_correct != ($#inputs + 1)) {
          $number_correct = 0;
          $iterations++;

          # Try flipping two rules
          $r1 = rand($#rules + 1);
          $r2 = rand($#rules + 1);
          @rules[$r1, $r2] = @rules[$r2, $r1];

          for ($i = 0; $i <= $#inputs; $i++) {
              # We'll start with the current input, and transform it
              $output = $inputs[$i];
              for ($r = 0; $r <= $#rules; $r++) {
                    $output =~ s/$rules[$r][0]/$rules[$r][1]/g;
```

```
                }
                # Now check answer against the "real" answer in the checkfile
                if ($output eq $answers[$i]) {
                     $number_correct++;
                }
            }
      }
      $total_iterations += $iterations;
      print "Trial $t took $iterations iterations\n";

  }

  # Now that we're done, the average iterations is the total over the number of trials
  $average_iterations = $total_iterations / $number_of_trials;
  printf "\nAfter $number_of_trials trials, the average solution time is %.2f iterations\n", $average_iterations;
```

Assignment 2: Due 9/23

1. The program `italian.pl` (presented in class) provides a possible (but extremely stupid and inefficient) approach to finding a rule ordering that is consistent with the data. In psuedo-code:

   ```
   Pick one rule (R1) at random;
   Pick a second rule (R2) at random;
   Swap R1 and R2 in the list of ordered rules;
   ```

   Can you think of a more sensible approach, that might guide the learner to modify the current hypothesis in a more efficient way? Explain your proposal in prose (≈1 paragraph) and try to formalize it in pseudo-code

   - Optional: try to implement your idea by modifying the *italian.pl* program. If your idea requires getting Perl to do something that we haven't seen before, ask me and I can try to point you to the relevant commands. (This exercise would be very helpful in cementing your new-found Perl skills, but it is not required, since I want to leave you time to do the readings)

2. Read Hutchinson, chapter 1, on basic terminology to characterize learning algorithms. Now consider the following learning agent: a phonology student, whose task is to find the solution to a typical phonology problem set. Characterize the learning task. What is the training set like? (§1.1; open/closed domain, clean/noisy, etc.) How would you characterize the data (§1.2)? What is the solution space (or what determines it)? What type of algorithm(s) do such agents tend to employ (§1.5)? Is it supervised? unsupervised? How do you know that a solution is right? Does order of examples play a role?

   Now think about children (infants) learning the phonology of their language. How does the task differ from that of a phonology student? Is there a difference between learning phonotactics (the inventory of the language, possible combinations) and learning alternations?